Matthew Duc Nguyen

dnguy521@ucr.edu

University of California, Riverside

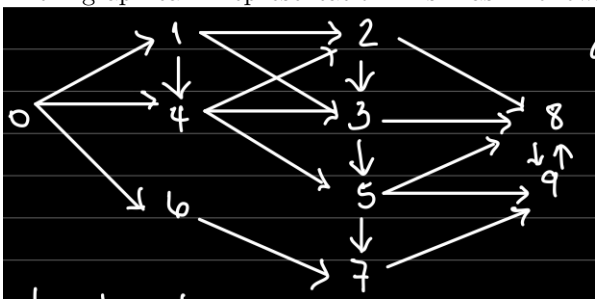
github: project2-matthewduc

1 Introduction

This report attempts to explain a correct implementation of parallel BFS with faster run time than sequential BFS. Due to constant "segmentation" and "bus error (core dumped)" with given input graphs under directory /usr/local/cs214/, the report shows correctness through an example graph in CSR format similar to input graphs. The example graph is similar to the one presented in lecture slides. The sequential BFS algorithm revisits the same vertices unnecessarily. The parallel BFS, visiting vertices in parallel and filtering out visited vertices, for a graph of 10 nodes compared to sequential BFS is 1.56x faster.

2 Graph

The directed graph in CSR format is as follows:
int offset[10] = 0,3,6,8,10,13,15,17,18,19;
int E[20] = 1,4,6,2,3,4,3,8,8,5,2,3,5,8,9,5,7,9,9,8;
The graphical representation is as follows:



[Figure 1]

3 Sequential BFS

From the initial commit of parallel BFS project, the sequential algorithm is as follows:

```
void seq_BFS(int n, int m, int* offset,
    int* E, int s, int* dist) {
    for (int i = 0; i < n; i++)
        dist[i] = -1;
    dist[s] = 0;
    int* q = new int[n];
    q[0] = s;
    int start = 0, end = 1;
    while (start != end) {
        int cur_node = q[start];
        int cur_dis = dist[cur_node];

        for (int i = offset[cur_node];
            i < offset[cur_node+1]; i++) {
            int v = E[i];
            if (dist[v] == -1) {
                dist[v] = cur_dis + 1;
                q[end++] = v;
            }
        }
        start++;
    }
    cout << "traversed: " << end
        << " vertices" << endl;
}
```

This algorithm maintains a $dist[]$ array initialized with -1 at each position. A queue $q[]$ inserts every neighbor of the current vertex into the queue. The distance is updated if the distance value for the vertex is -1 ($if(dist[v] == -1)$; the distance value of the vertex closer to the source is incremented by 1 and the vertex is placed into the queue. This sequential algorithm is correct in calculating the distance from the source node. The distance from output of the graph of [Figure 1] is v Distance:0 1 2 2 1 2 1 2 3 3 with vertex 0 as the source vertex. However, the time is slow and can be parallelized.

4 Parallel BFS

From [Figure 1], vertex 0 would first visit vertices 1, 4, and 6. The distance to these three vertices is 1. The neighbors of the 1, 4, and 6 can be visited in parallel. The effective neighbors, those that have not been visited, of these vertices are added too queue and visited. The following functions describes the parallel implementation.

4.1 Prefix Sum Function

```
int scan_down(int*A, int*B, int* LS,
        int n, int offset){
   if(n==1){
       B[0] = offset + A[0];
       return 0;
   }
   int m = n/2;
   int L, R;
   L = cilk_spawn
    scan_down(A, B, LS, m, offset);
   R = scan_down(A+m, B+m, LS+m,
    n-m, offset+LS[m-1]);
   cilk_sync;
   return L+R;
}

int scan_up(int* A, int* LS, int n){
   if(n==1) return A[0];
   int m = n/2;
   int L, R;
   L = cilk_spawn scan_up(A, LS, m);
   R = scan_up(A+m, LS+m, n-m);
   LS[m-1]=L;
   cilk_sync;
   return L+R;
}

void scan(int* A, int* B, int n){
    int* LS = new int[n-1];
    scan_up(A, LS, n);
    scan_down(A, B, LS, n, 0);
}
```

The scan function implemented makes use of a left sum array to compute the sum of previous values in an array. The first scan-up function computes the leftsum array. The second scan-down array computes the prefix sum the remaining elements and store in array B.

4.2 Filter Function

```
void Filter(int* A, int n,
        int* B, int* flagsA){
    int* ps = new int[n];
    cilk_for(int i=0;i<n;i++){
        ps[i] = 0;
    }
    scan(flagsA, ps, n);
    cilk_for (int i = 0; i<n;i++){
        B[ps[i]-1] = A[i];
    }
}
```

The filter function implemented takes a pointer to array A, and integer n, a pointer to array B, and a point of an array of Flags. The function calls scan to compute and prefix sums, this is then used as positions to store values of array A that are filtered.

4.3 Parallel BFS Function

```
void BFS(int n, int m, int* offset,
        int* E, int s, int* dist) {
    cilk_for(int i =0; i<n; i++)
        dist[i] = -1;
    dist[s] = 0;

    bool* visited = new bool[n];
    cilk_for (int i = 0; i < n; i++)
        visited[i] = false;

    int* frontier = new int[n];
    cilk_for (int i = 0; i < n; i++)
        frontier[i] = -1;

    frontier[0] = s;

    // while |frontier|
    int start = 0; int end = 1;
    while(start!=end){
        if (end == n) end--;
        frontier = next_Frontier(n, m, offset, E,
        frontier, dist, visited, start, end);
    }
}
```

Integers start and end track the number of vertices visited and inserted into the queue, respectively. A new frontier of effective neighbors is generated with the next frontier function while the $start! = end$.

4.4 Next Frontier Function

```
int* next_Frontier(int n, int m,
int offsetCount, newFrontCount = 0;
    // cilk_for x in current frontier
    cilk_for(int x=start; x<end; x++){
    int current_node = frontier[x];
    if(current_node!=-1) {
    if (start<n) start++;
    int out_neighbors =
    offset[current_node+1]
        -offset[current_node];
    int* neighbors = new int[out_neighbors];
    int* effective_neighbors =
        new int[out_neighbors];
    int* neighbors_flag =
        new int[out_neighbors];

    for(int i=0;i<out_neighbors;i++){
        effective_neighbors[i] = -1;
        neighbors_flag[i] = 0;
        int v = E[offset[current_node]+i];
        neighbors[i] = v;
        if((!visited[v]) && CAS(&visited[v],
            false,true)){
                neighbors_flag[i] = 1;
            int cur_dist =
                dist[current_node];
            if (dist[v] == -1) {
            dist[v] = cur_dist + 1;
                }
            }
        }
    Filter(neighbors, out_neighbors,
    effective_neighbors, neighbors_flag);
    cilk_for(int i = 0;i<out_neighbors;i++){
    if(effective_neighbors[i] != -1){
        frontier[end++] =
        effective_neighbors[i];
                }
            }
        }
    }
    return frontier;
}
```

This next frontier function only generates parallel for loops from *start* to *end*. Within each parallel task, the vertex considers its neighbors. Each neighbor is flagged if it has not been visited in the graph. These flags along with the neighbors of the vertex are passed to the filter function to return the effective neighbors of the vertex. Each effective neighbor is added to the queue. The control is similar to the the sequential algorithm with queue size, differing with the count of the number of the vertices visited and inserted into the queue being tracked.

4.5 Granularity Control

The most notable granularity control is within the next frontier function:

```
cilk_for(int x=start; x<end; x++)
```

This parallel for loop only creates parallel tasks based on the start and end "pointers" to the queue. This prevents the start of unnecessary parallel processes.

5 Conclusion

Computing the distances of the graph in [Figure 1] using the sequential and parallel BFS algorithms yielded the following runtimes in μ seconds:

```
traversed: 10 vertices
sequential time: 2.21729
BFS, Round 0, time: 1.71661e-05
BFS, Round 1, time: 1.38283e-05
BFS, Round 2, time: 1.28746e-05
BFS, Round 3, time: 1.28746e-05
BFS, Round 4, time: 1.40667e-05
BFS, Average time: 1.41621
```

[Figure 2]

The sequential runtime is 2.21 μ seconds while the parallel runtime is 1.41 μ seconds. This is a 1.567 times speedup.

6 Acknowledgements

CS214 has been one of the most chellenging courses I have taken. I have learned a lot more about C++, running code in parallel, and learning about theory of time bounds and complexities. I hope to parallelize more program in with the knowledge learned from this course. Thank you for a very enlightening quarter. I am using 2 grace days along with this submission because week 10 is very exhausting.